# Gaining Insight into Parallel Program Performance using HPCToolkit

John Mellor-Crummey
Department of Computer Science
Rice University

**http://hpctoolkit.org**

# Challenges for Computational Scientists

- Rapidly evolving platforms and applications
  - architecture
    - rapidly changing multicore microprocessor designs
    - increasing architectural diversity
      CPU, GPU, APU, manycore (e.g., Xeon Phi)
    - increasing scale of parallel systems
  - applications
    - augment computational capabilities
- Computational scientist needs
  - adapt to changes in emerging architectures
    - adding threading and/or offloading to accelerators
  - improve scalability within and across nodes
  - assess weaknesses in algorithms and their implementations

**Performance tools can play an important role as a guide**

# Performance Analysis Challenges

- Complex node architectures are hard to use efficiently
  - multi-level parallelism: multiple cores, ILP, SIMD, accelerators
  - multi-level memory hierarchy
  - result: gap between typical and peak performance is huge
- Complex applications present challenges
  - measurement and analysis
  - understanding behaviors and tuning performance
- Supercomputer platforms compound the complexity
  - unique hardware & microkernel-based operating systems
  - multifaceted performance concerns
    - computation
    - data movement
    - communication
    - I/O

# What Users Want

- Easy-to-use multi-platform, programming model independent tools
- Accurate measurement of complex parallel codes
  - large, multi-lingual programs
  - (heterogeneous) parallelism within and across nodes
  - optimized code: loop optimization, templates, inlining
  - binary-only libraries, sometimes partially stripped
  - complex execution environments
    - dynamic binaries on clusters; static binaries on supercomputers
    - batch jobs
- Effective performance analysis
  - insightful analysis that pinpoints and explains problems
    - correlate measurements with code for actionable results
    - support analysis at the desired level
      - intuitive enough for application scientists and engineers
      - detailed enough for library developers and compiler writers
- Scalable to petascale and beyond

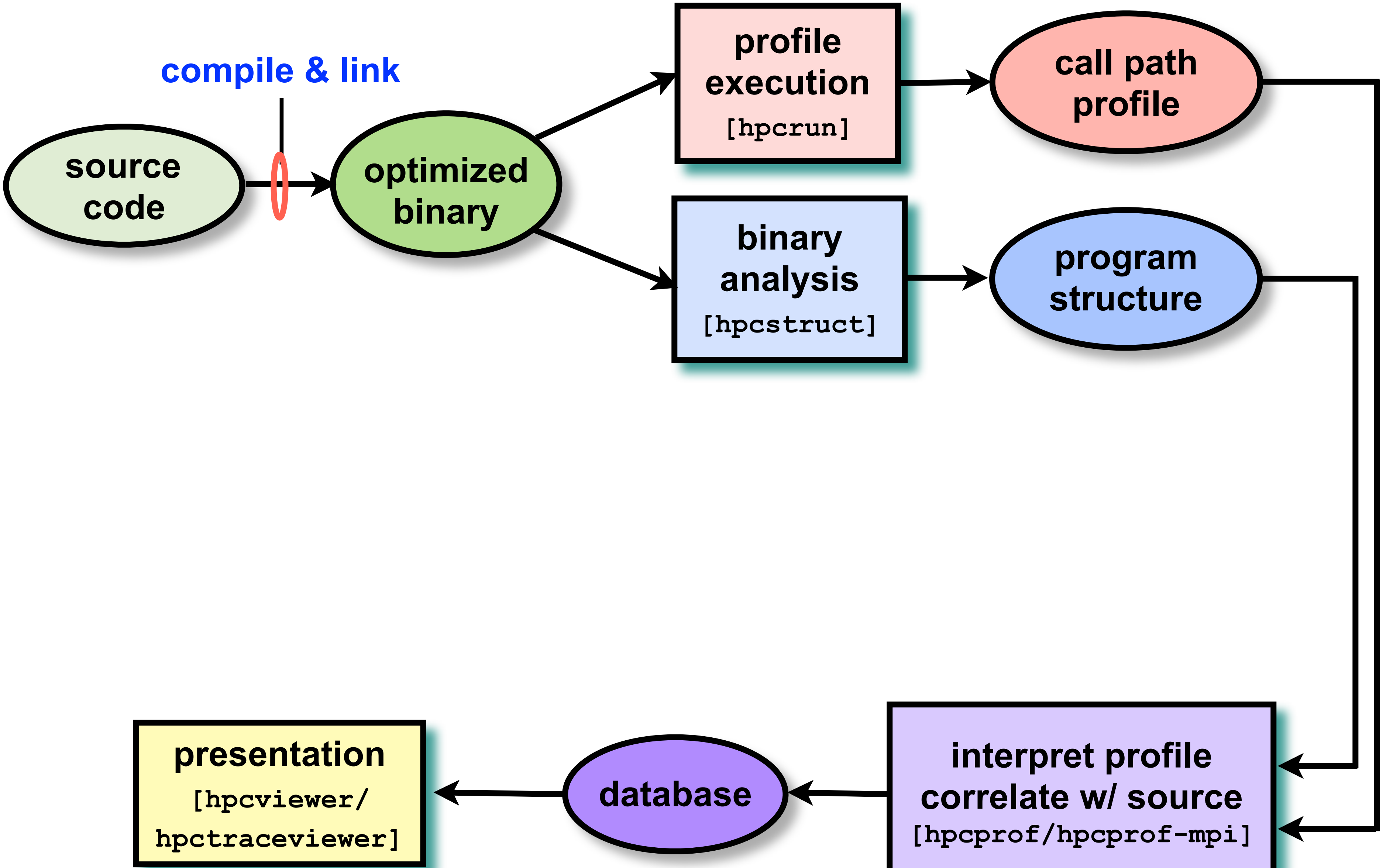
# Rice University's HPCToolkit

- Employs binary-level measurement and analysis
  - observe executions of optimized code
  - support multi-lingual codes with external binary-only libraries
- Uses sampling-based measurement (avoid instrumentation)
  - controllable overhead
  - minimize systematic error and avoid blind spots
  - enable data collection for large-scale parallelism
- Collects and correlates multiple derived performance metrics
  - diagnosis typically requires more than one species of metric
- Associates metrics with both static and dynamic context
  - loop nests, procedures, inlined code, calling context
- Supports top-down performance analysis
  - natural approach that minimizes burden on developers

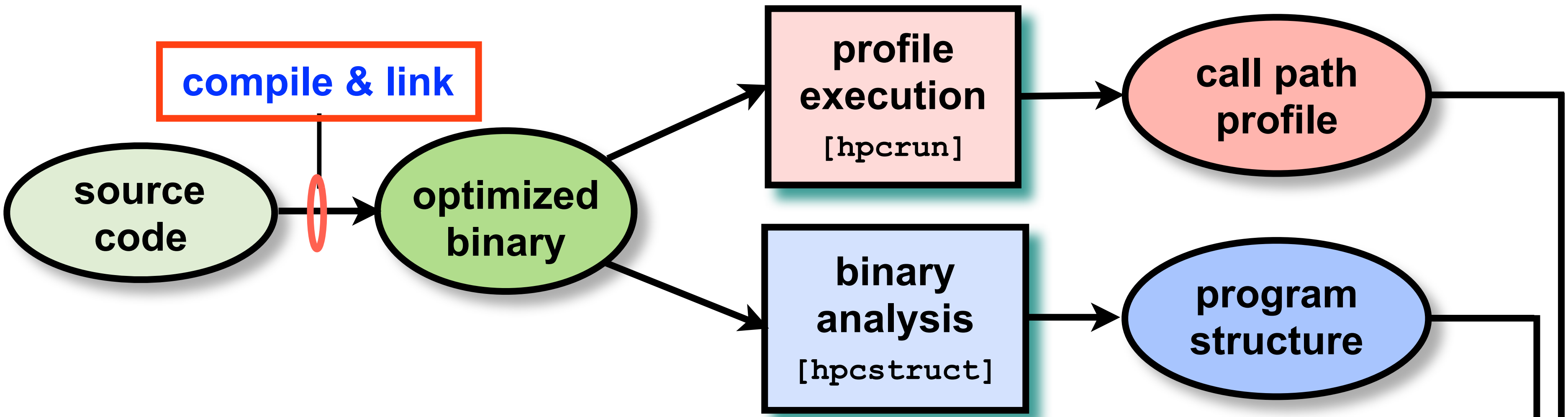
# Outline

- **Overview of Rice's HPCToolkit**
- **Pinpointing scalability bottlenecks**
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- **Understanding temporal behavior**
- **Assessing process variability**
- **Understanding threading, GPU, and memory hierarchy**
  - — blame shifting
  - — attributing memory hierarchy costs to data
- **Summary and challenges ahead**

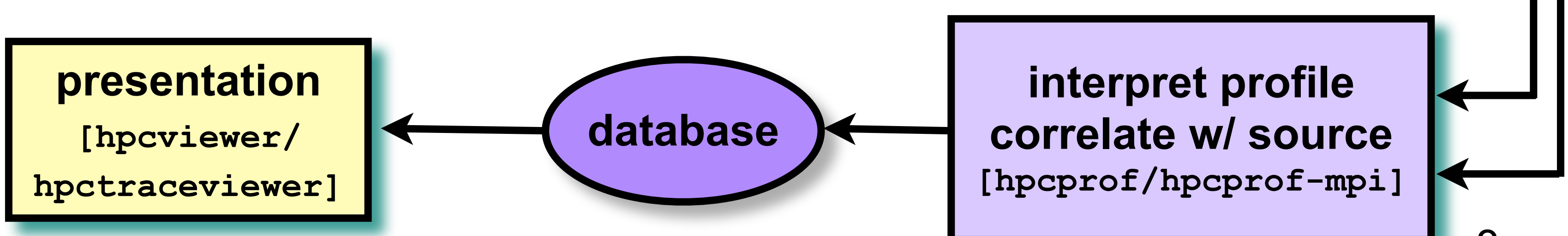
# HPCToolkit Workflow



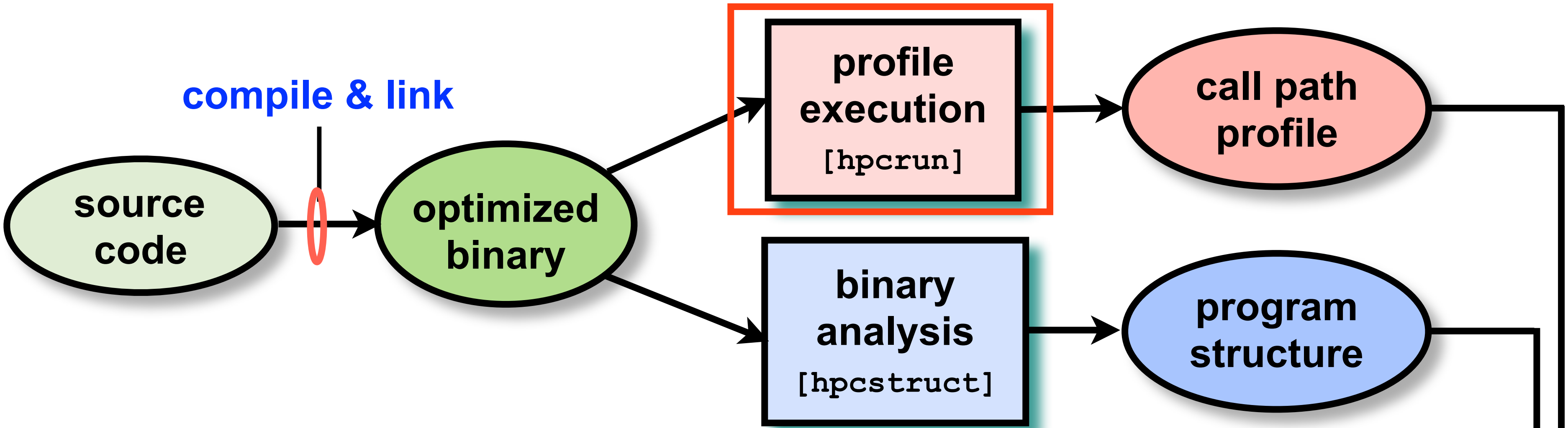
# HPCToolkit Workflow




- For dynamically-linked executables, e.g., Linux
  — compile and link as you usually do
- For statically-linked executables, e.g., Blue Gene/Q
  — add monitoring by using `hpclink` as prefix to your link line

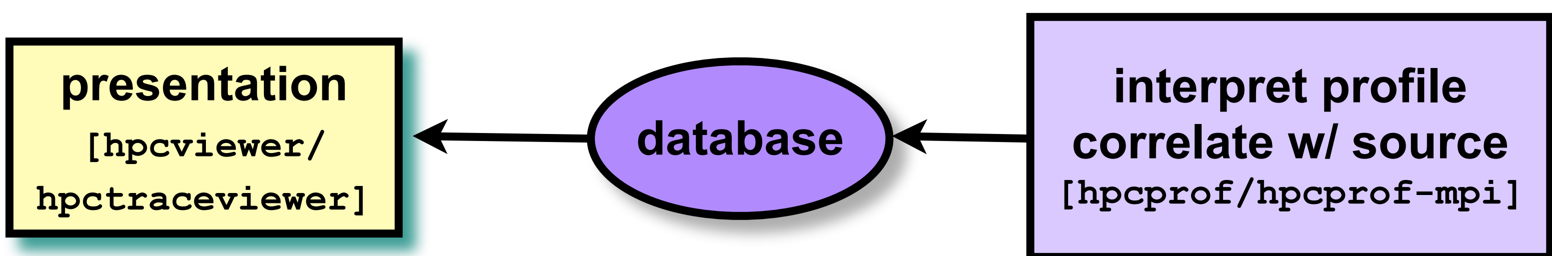


# HPCToolkit Workflow




- **Measure execution unobtrusively**
  - launch optimized application binaries
    - dynamically-linked applications: launch with hpcrun

      e.g., mpirun -np 8192 hpcrun -t -e WALLCLOCK@5000 flash3 ...
    - statically-linked applications: control with environment variables
  - collect statistical call path profiles of events of interest

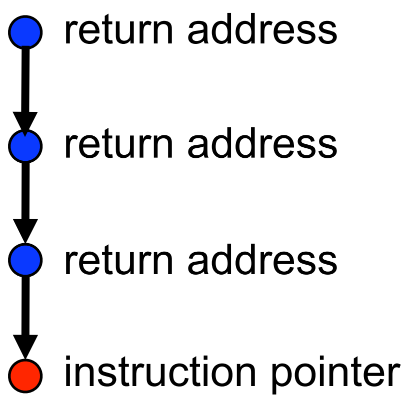


# Call Path Profiling

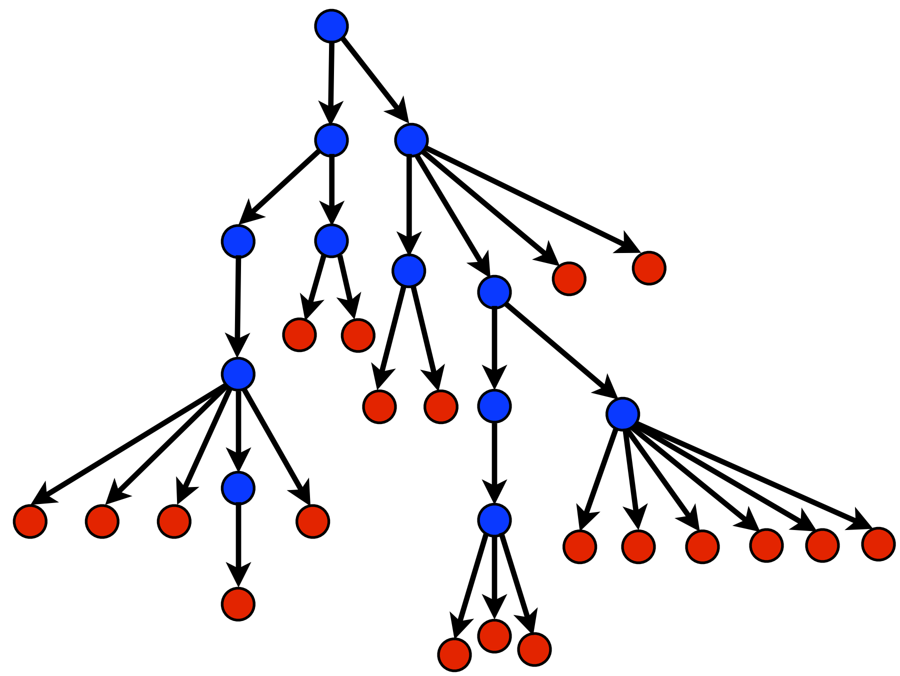
**Measure and attribute costs in context**

- **sample timer or hardware counter overflows**
- **gather calling context using stack unwinding**

Call path sample

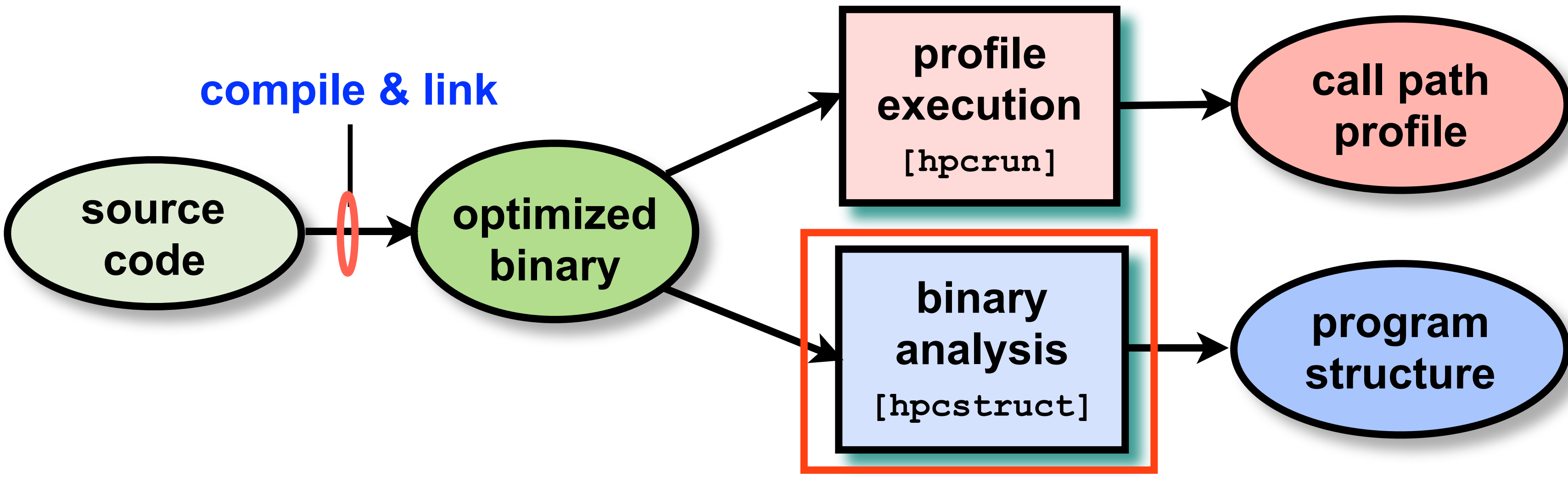



Calling context tree

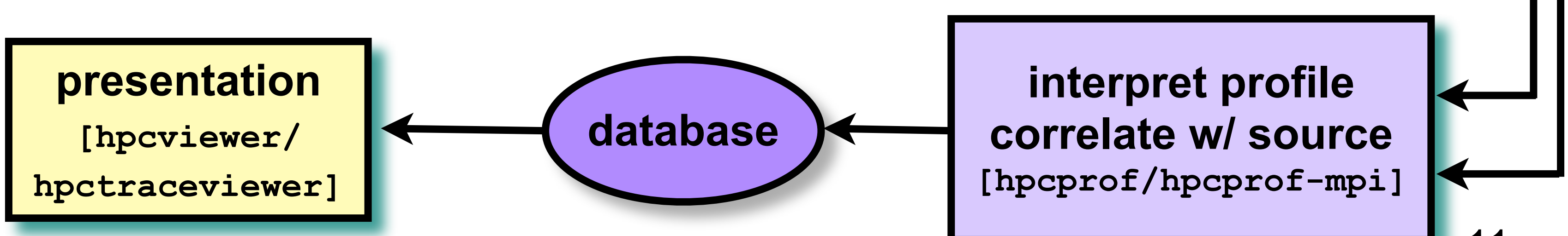


**Overhead proportional to sampling frequency...**
**...not call frequency**

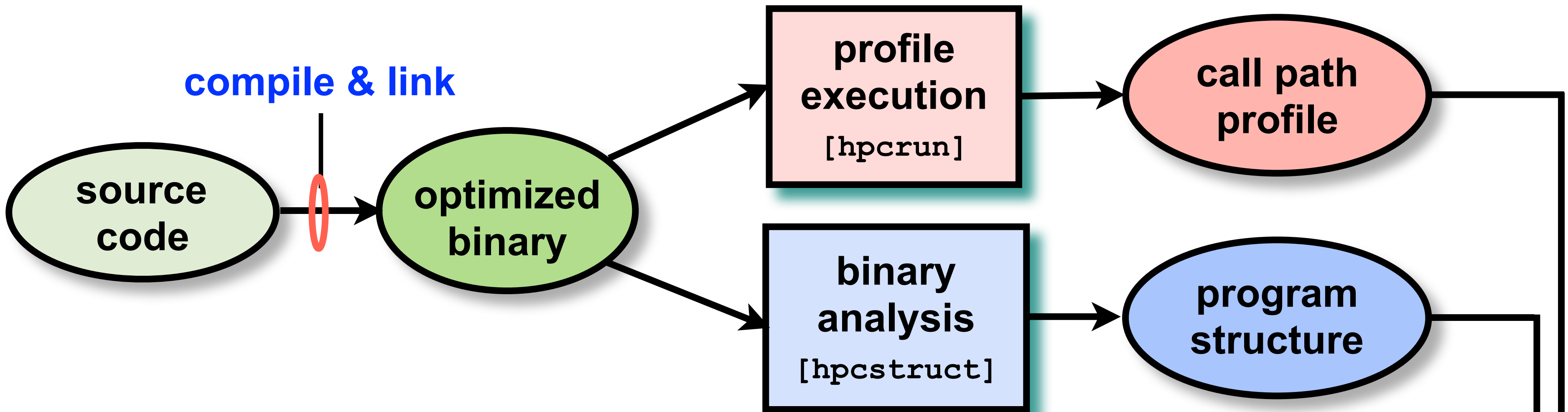
# HPCToolkit Workflow



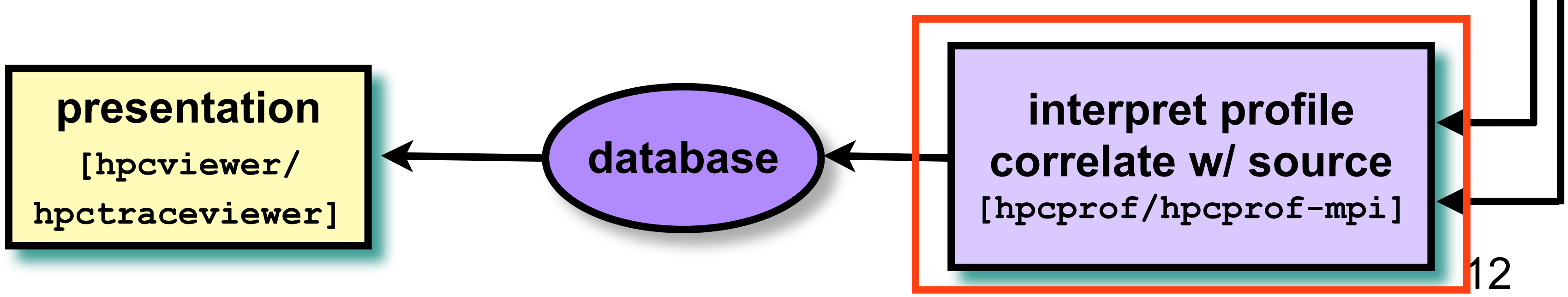

- Analyze binary with `hpcstruct`: recover program structure
  - — analyze machine code, line map, debugging information
  - — extract loop nesting & identify inlined procedures
  - — map transformed loops and procedures to source



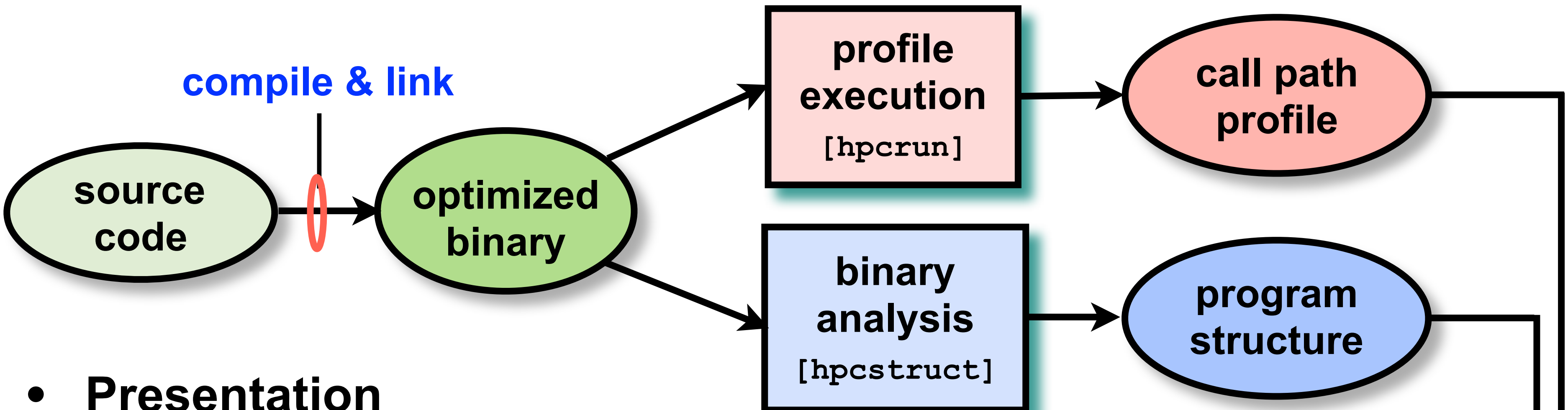
# HPCToolkit Workflow



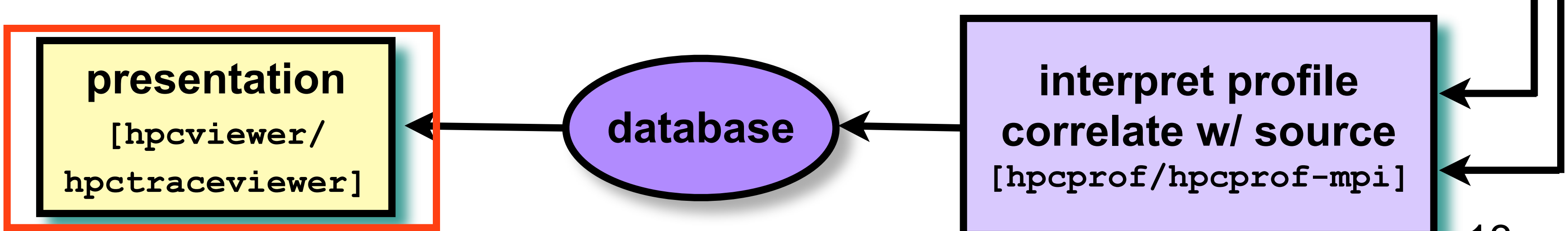

- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate metrics to static & dynamic program structure



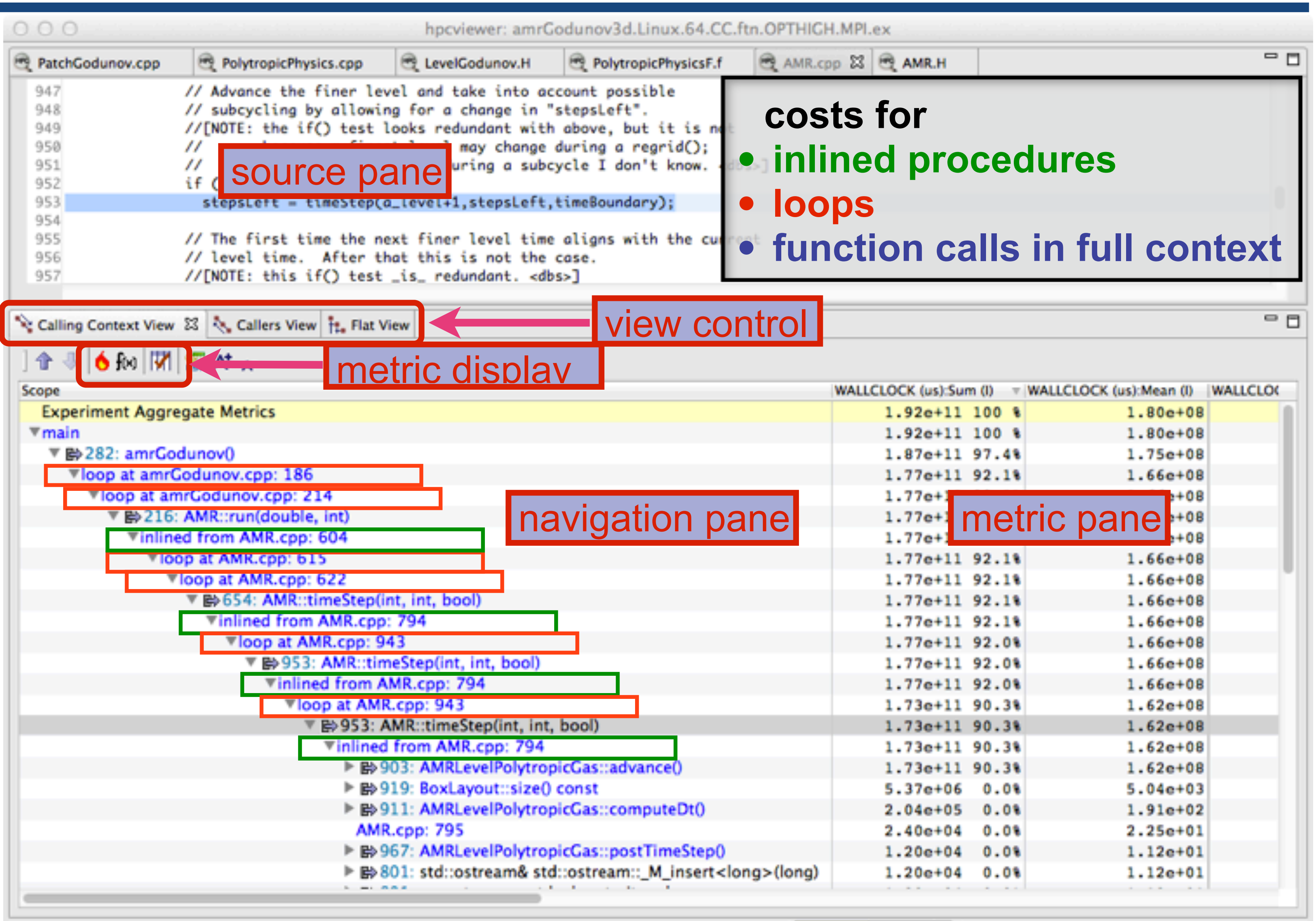
# HPCToolkit Workflow




- **Presentation**
  - — explore performance data from multiple perspectives
    - – rank order by metrics to focus on what's important
    - – compute derived metrics to help gain insight
      - e.g. scalability losses, waste, CPI, bandwidth
  - — graph thread-level metrics for contexts
  - — explore evolution of behavior over time



# Analyzing Chombo@1024 cores with hpcviewer



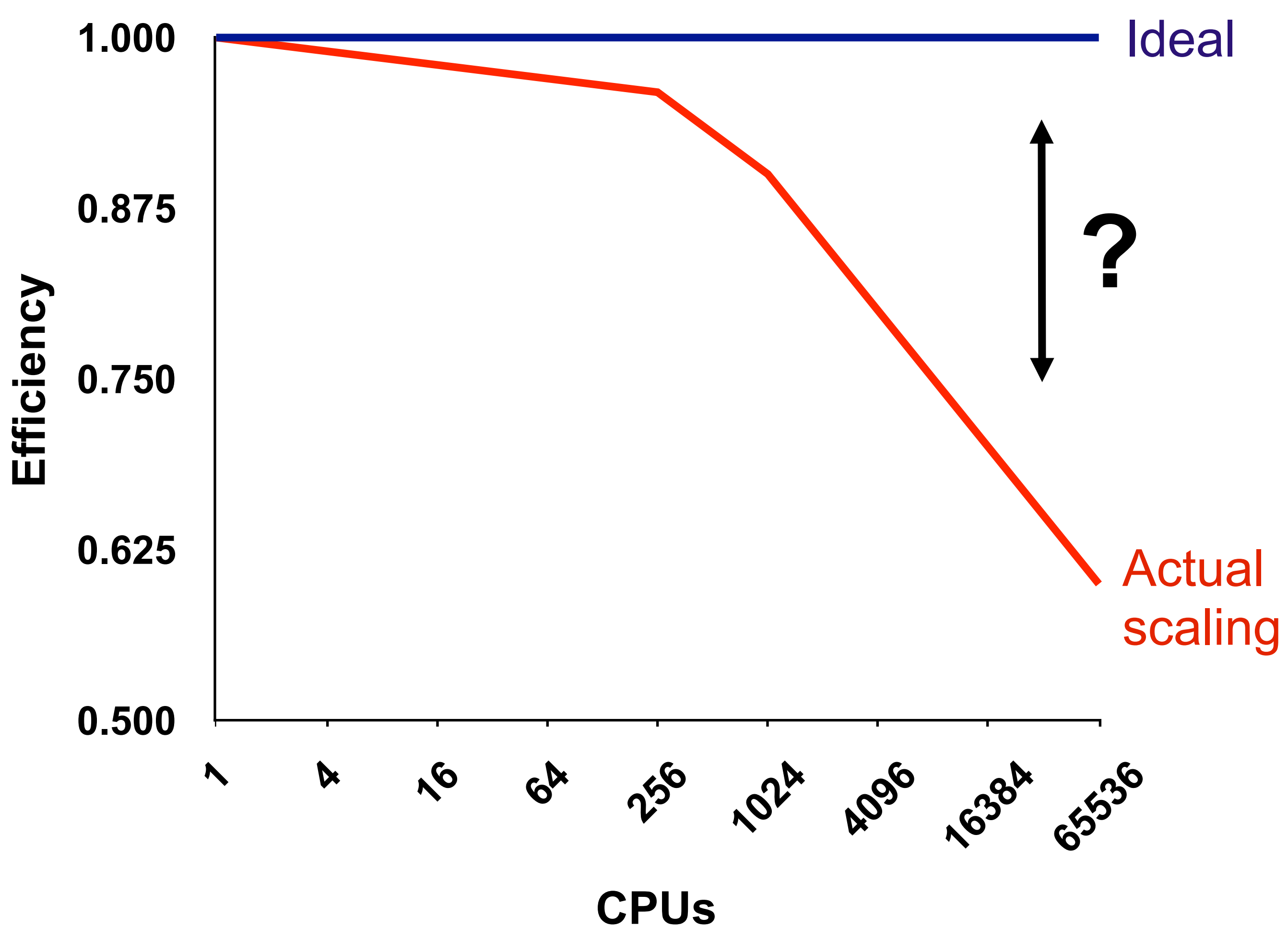
# Principal Views

- **Calling context tree view - "top-down" (down the call chain)**
  - associate metrics with each dynamic calling context
  - high-level, hierarchical view of distribution of costs
  - example: quantify initialization, solve, post-processing
- **Caller's view - "bottom-up" (up the call chain)**
  - apportion a procedure's metrics to its dynamic calling contexts
  - understand costs of a procedure called in many places
  - example: see where PGAS put traffic is originating
- **Flat view - ignores the calling context of each sample point**
  - aggregate all metrics for a procedure, from any context
  - attribute costs to loop nests and lines within a procedure
  - example: assess the overall memory hierarchy performance within a critical procedure

# Outline

- Overview of Rice's HPCToolkit
- **Pinpointing scalability bottlenecks**
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  - — blame shifting
  - — attributing memory hierarchy costs to data
- Summary and challenges ahead

# The Problem of Scaling



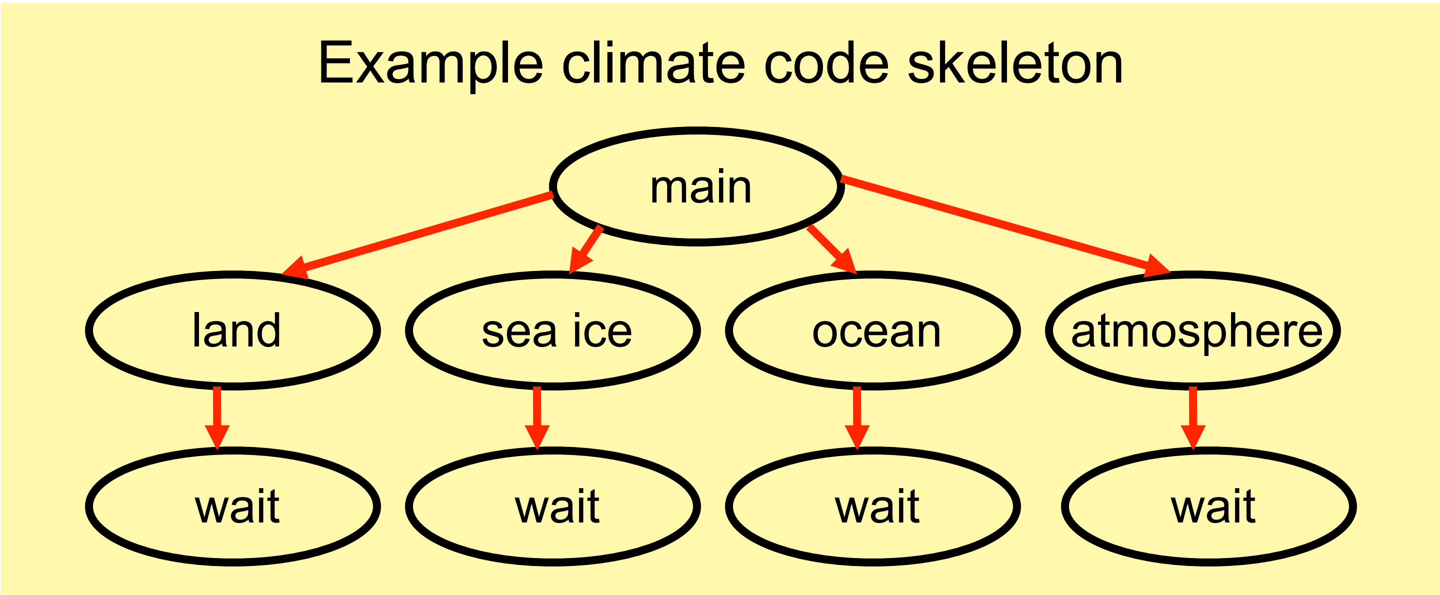

Note: higher is better

# Wanted: Scalability Analysis

- Isolate scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem

# Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**
  - — **modern software uses layers of libraries**
  - — **performance is often context dependent**



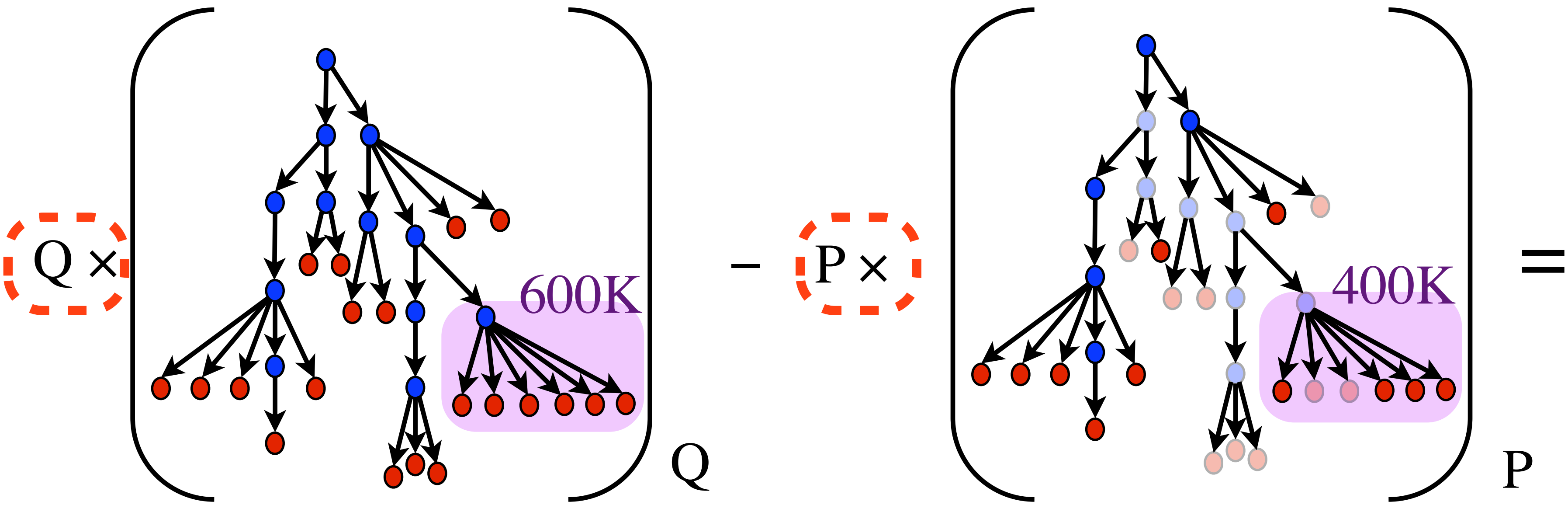

- **Monitoring**
  - — **bottleneck nature: computation, data movement, synchronization?**
  - — **2 pragmatic constraints**
    - – **acceptable data volume**
    - – **low perturbation for use in production runs**

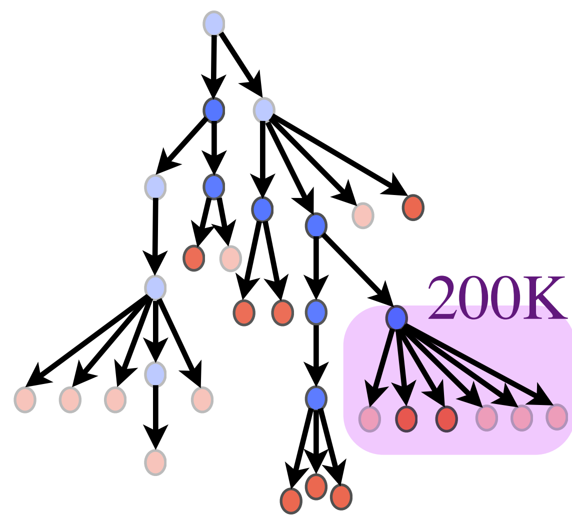
# Performance Analysis with Expectations

- You have performance expectations for your parallel code
  - strong scaling: linear speedup
  - weak scaling: constant execution time

- Put your expectations to work
  - measure performance under different conditions
    - e.g. different levels of parallelism and/or different problem size
  - express your expectations as an equation
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively

# Pinpointing and Quantifying Scalability Bottlenecks

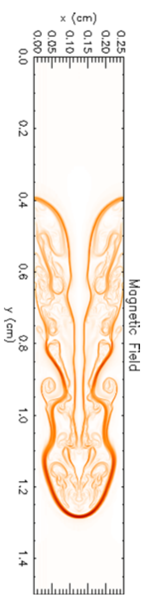


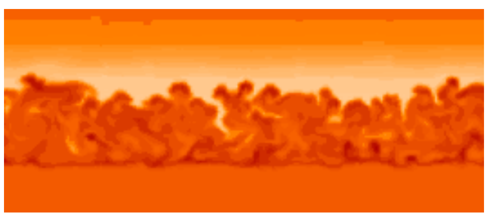

coefficients for analysis
of strong scaling

# Scalability Analysis Demo: FLASH3

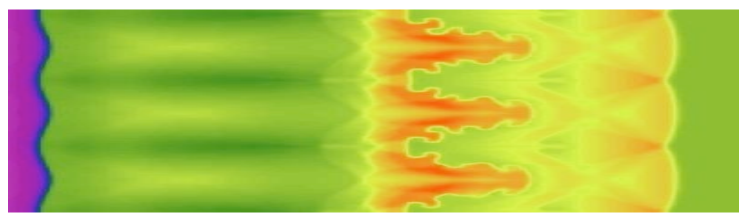
| | |
|---|---|
| **Code:** | **University of Chicago FLASH3** |
| **Simulation:** | **white dwarf detonation** |
| **Platform:** | **Blue Gene/P** |
| **Experiment:** | **8192 vs. 256 processors** |
| **Scaling type:** | **weak** |



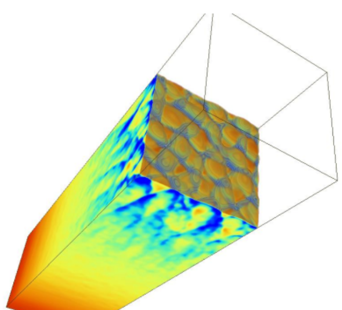

*Magnetic Rayleigh-Taylor*



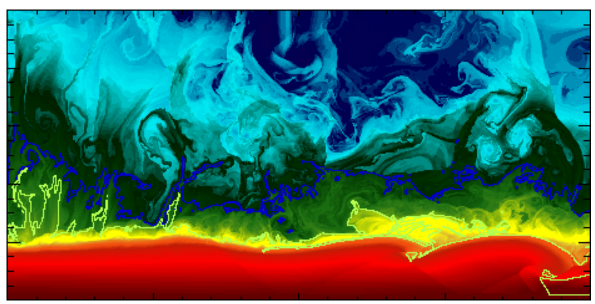
*Nova outbursts on white dwarfs*



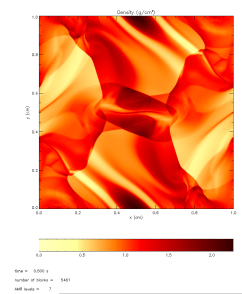
*Laser-driven shock instabilities*



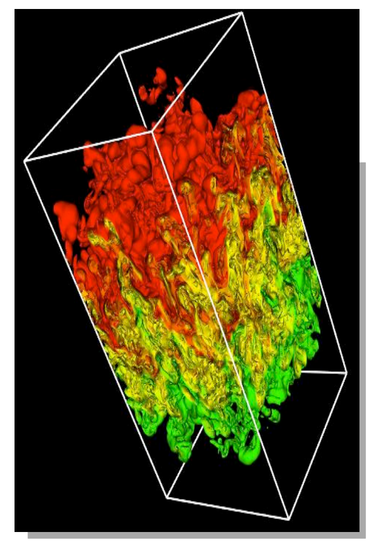
*Cellular detonation*



*Helium burning on neutron stars*



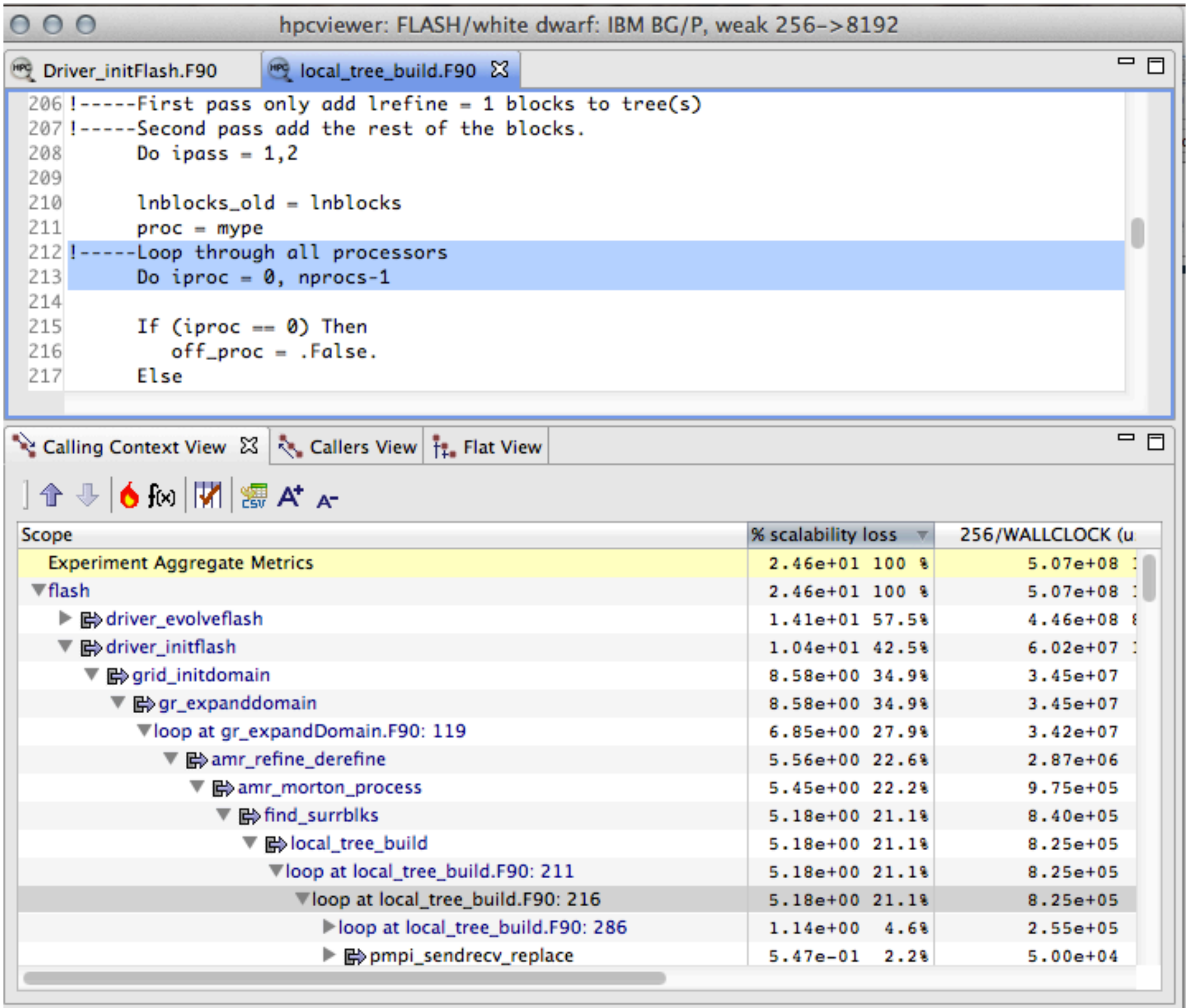
*Orzag/Tang MHD vortex*



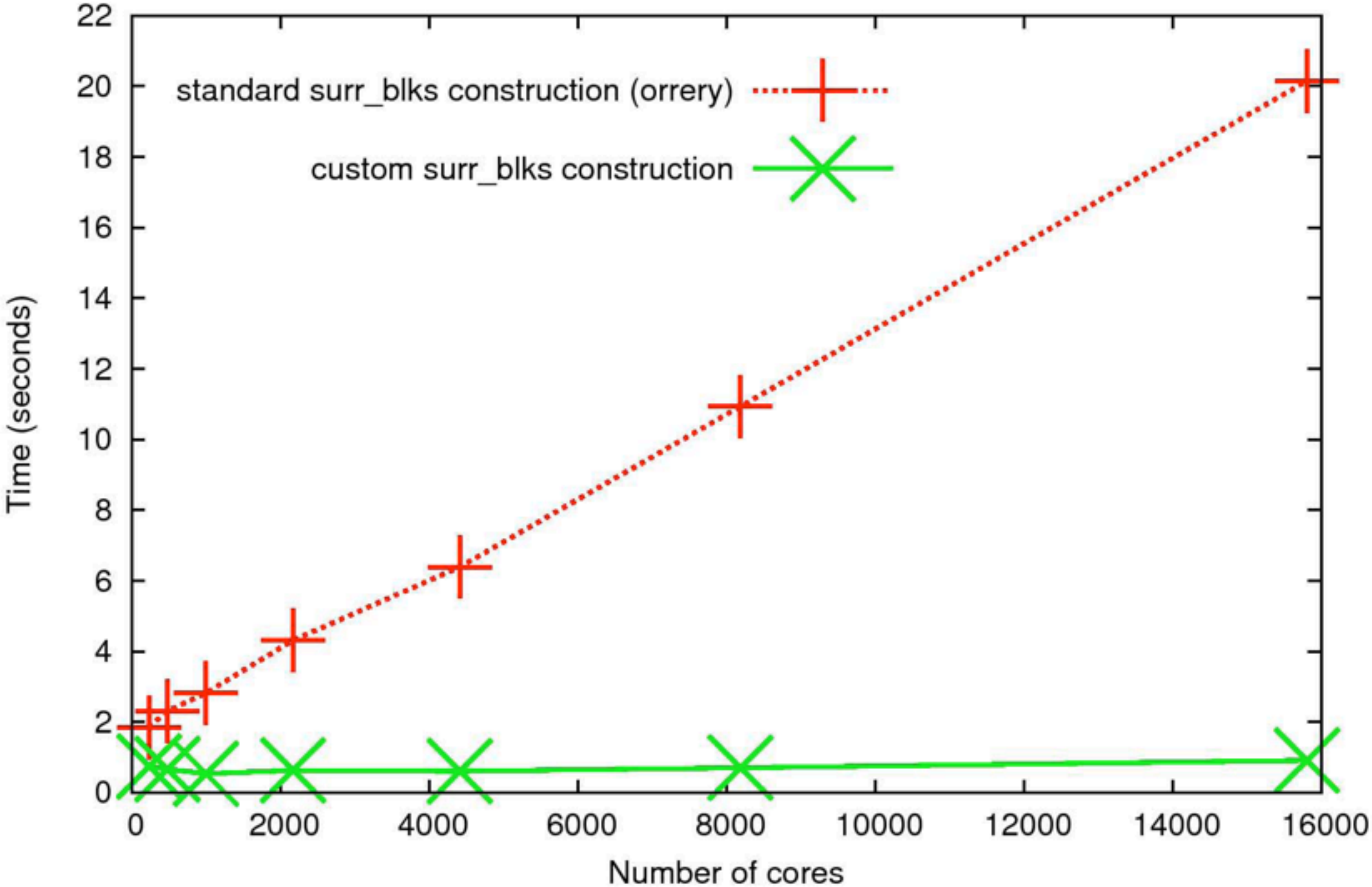
*Rayleigh-Taylor instability*

Figures courtesy of FLASH Team, University of Chicago

# Scalability Analysis of Flash3 (Demo)



# Improved Flash Scaling of AMR Setup

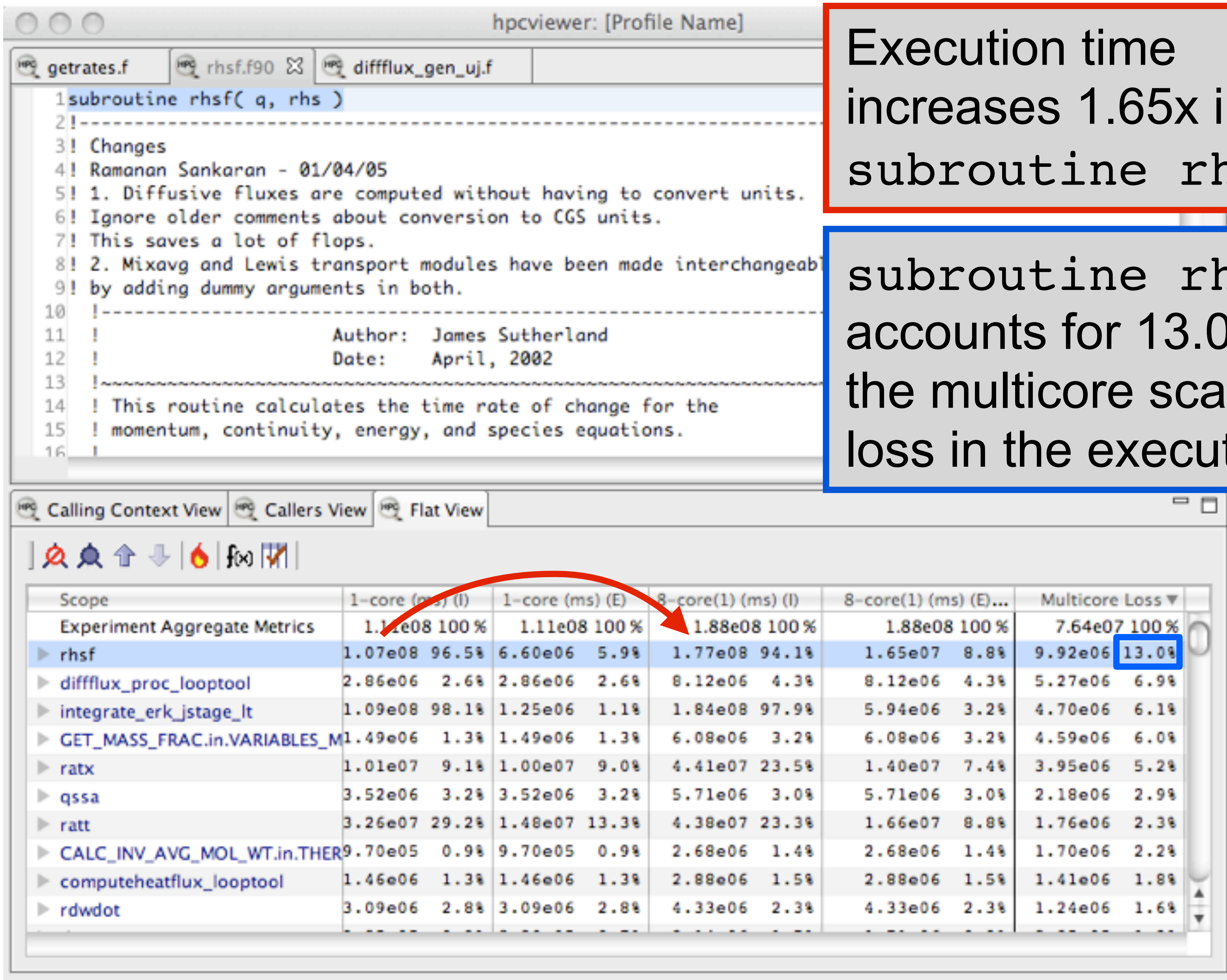



Graph courtesy of Anshu Dubey, U Chicago

# Scaling on Multicore Processors

- **Compare performance**
  - **single vs. multiple processes on a multicore system**
- **Strategy**
  - **differential performance analysis**
    - **subtract the calling context trees as before, unit coefficient for each**

# S3D: Multicore Losses at the Procedure Level

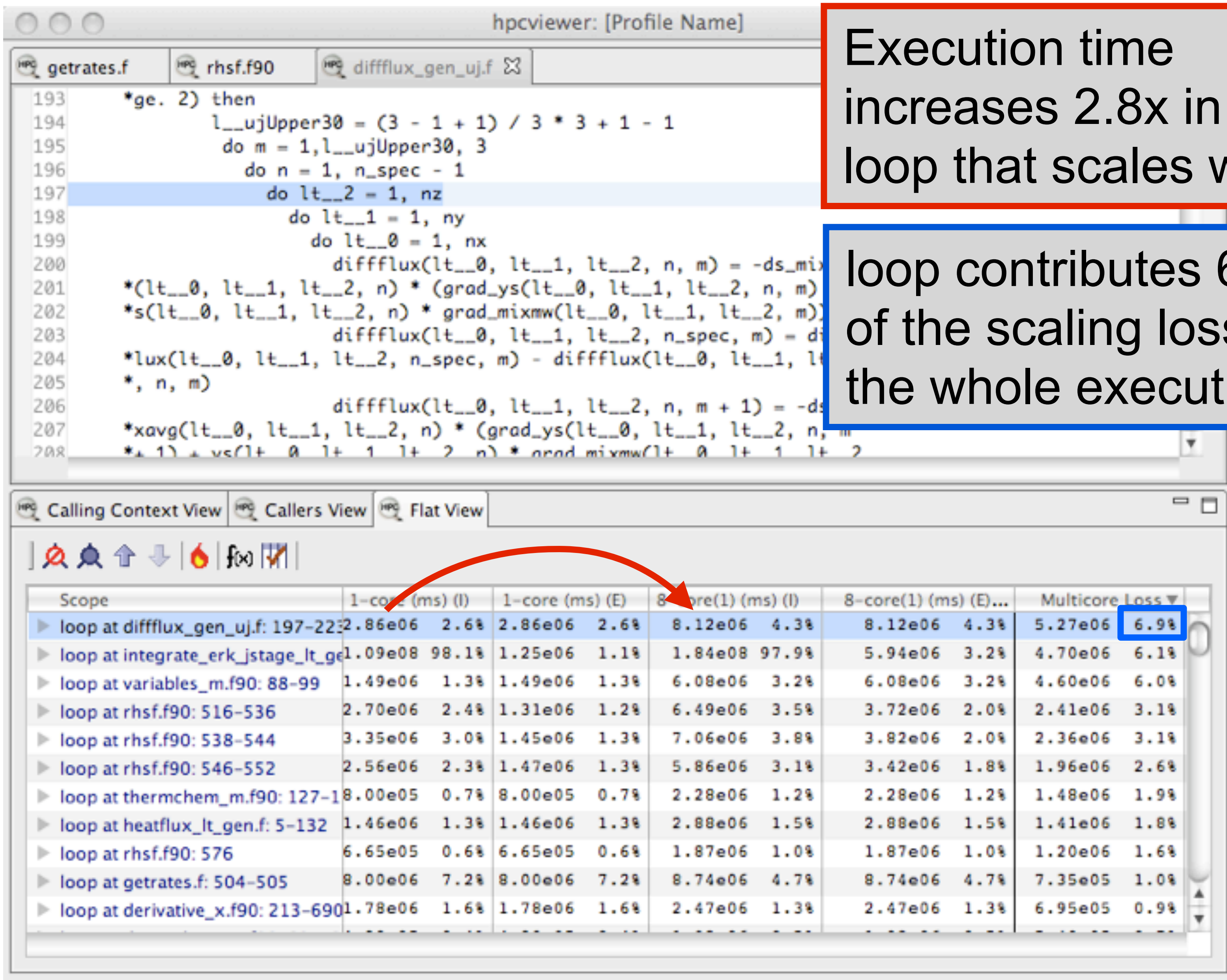



| Scope | 1-core (ms) (I) | 1-core (ms) (E) | 8-core(1) (ms) (I) | 8-core(1) (ms) (E)... | Multicore Loss ▼ |
|---|---|---|---|---|---|
| Experiment Aggregate Metrics | 1.11e08 100 % | 1.11e08 100 % | 1.88e08 100 % | 1.88e08 100 % | 7.64e07 100 % |
| rhsf | 1.07e08 96.5% | 6.60e06 5.9% | 1.77e08 94.1% | 1.65e07 8.8% | 9.92e06 13.0% |
| diffflux_proc_looptool | 2.86e06 2.6% | 2.86e06 2.6% | 8.12e06 4.3% | 8.12e06 4.3% | 5.27e06 6.9% |
| integrate_erk_jstage_lt | 1.09e08 98.1% | 1.25e06 1.1% | 1.84e08 97.9% | 5.94e06 3.2% | 4.70e06 6.1% |
| GET_MASS_FRAC.in.VARIABLES_M | 1.49e06 1.3% | 1.49e06 1.3% | 6.08e06 3.2% | 6.08e06 3.2% | 4.59e06 6.0% |
| ratx | 1.01e07 9.1% | 1.00e07 9.0% | 4.41e07 23.5% | 1.40e07 7.4% | 3.95e06 5.2% |
| qssa | 3.52e06 3.2% | 3.52e06 3.2% | 5.71e06 3.0% | 5.71e06 3.0% | 2.18e06 2.9% |
| ratt | 3.26e07 29.2% | 1.48e07 13.3% | 4.38e07 23.3% | 1.66e07 8.8% | 1.76e06 2.3% |
| CALC_INV_AVG_MOL_WT.in.THER | 9.70e05 0.9% | 9.70e05 0.9% | 2.68e06 1.4% | 2.68e06 1.4% | 1.70e06 2.2% |
| computeheatflux_looptool | 1.46e06 1.3% | 1.46e06 1.3% | 2.88e06 1.5% | 2.88e06 1.5% | 1.41e06 1.8% |
| rdwdot | 3.09e06 2.8% | 3.09e06 2.8% | 4.33e06 2.3% | 4.33e06 2.3% | 1.24e06 1.6% |

Execution time increases 1.65x in `subroutine rhsf`

`subroutine rhsf` accounts for 13.0% of the multicore scaling loss in the execution

# S3D: Multicore Losses at the Loop Level




Execution time increases 2.8x in the loop that scales worst

loop contributes 6.9% of the scaling loss for the whole execution

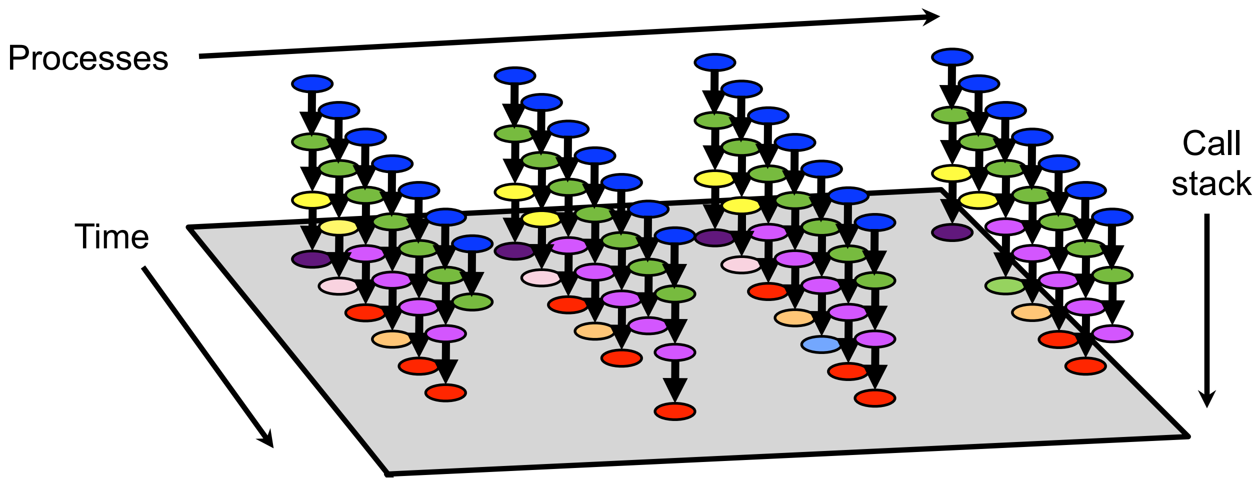
# Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and challenges ahead

# Understanding Temporal Behavior

- **Profiling compresses out the temporal dimension**
  - **—temporal patterns, e.g. serialization, are invisible in profiles**
- **What can we do? Trace call path samples**
  - **—sketch:**
    - **N times per second, take a call path sample of each thread**
    - **organize the samples for each thread along a time line**
    - **view how the execution evolves left to right**
    - **what do we view?**

      **assign each procedure a color; view a depth slice of an execution**

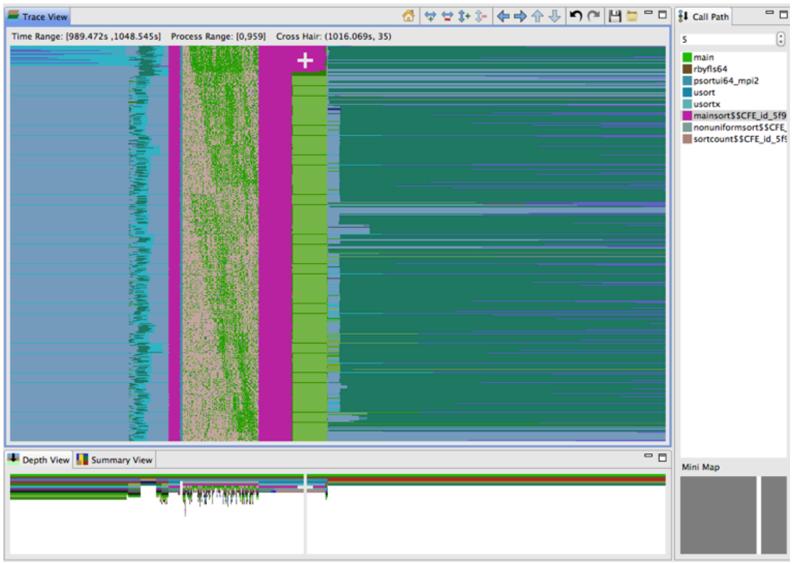


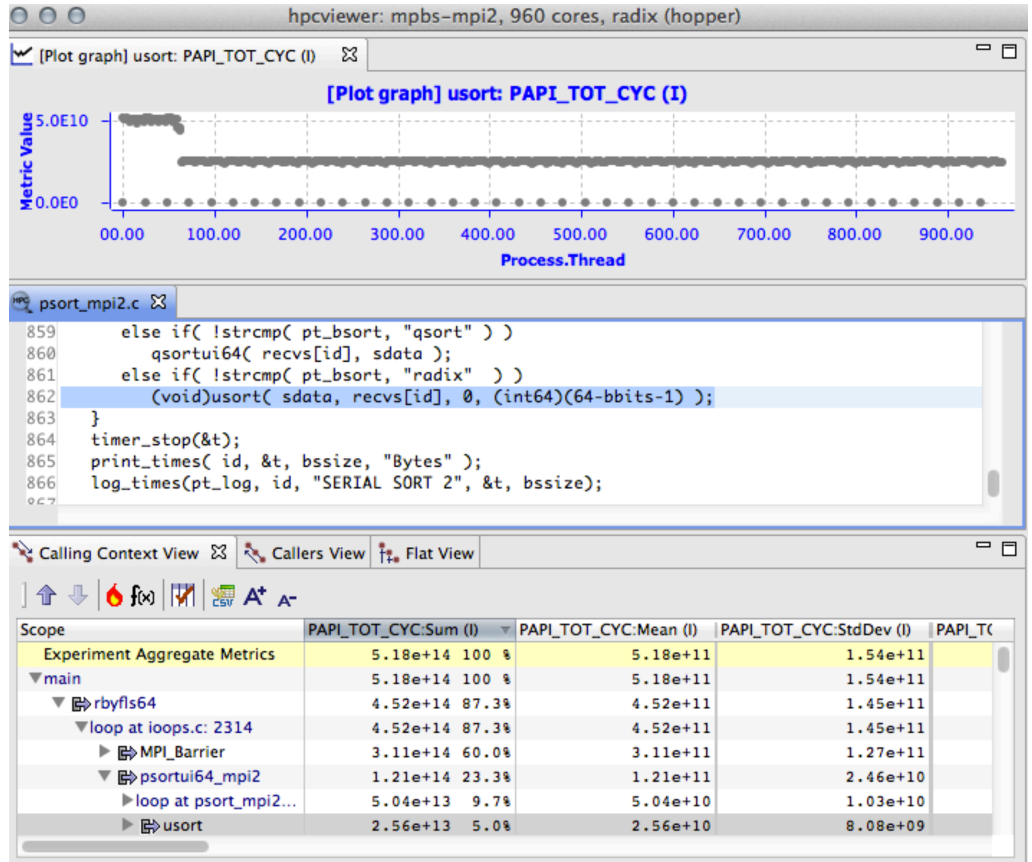
# Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
- Summary and challenges ahead

# MPBS @ 960 cores, radix sort

**Two views of load imbalance since not on a $2^k$ cores**



# Outline

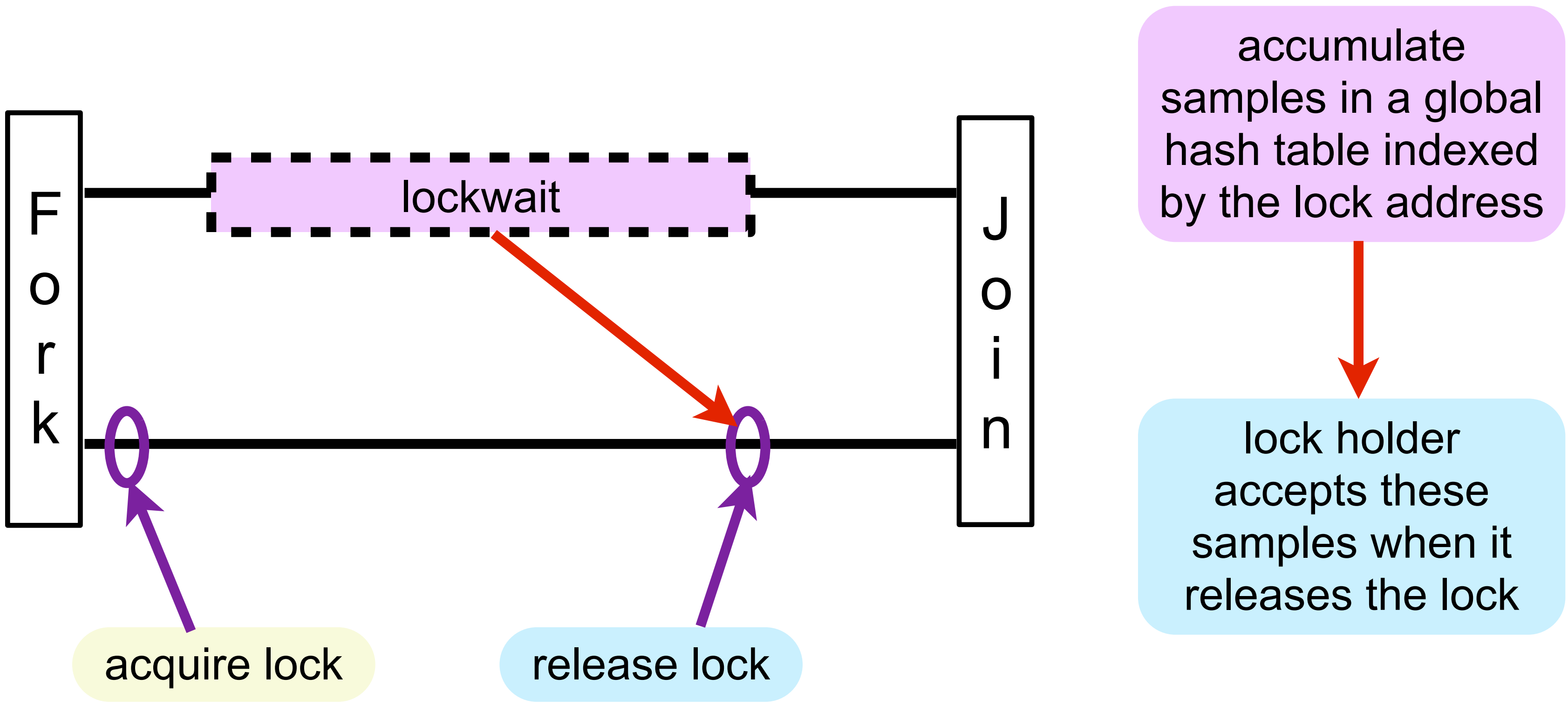
- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  - — blame shifting
  - — attributing memory hierarchy costs to data
- Summary and challenges ahead

# Blame Shifting

- **Problem: in many circumstances sampling measures symptoms of performance losses rather than causes**
  - worker threads waiting for work
  - threads waiting for a lock
  - MPI process waiting for peers in a collective communication
  - idle GPU waiting for work
- **Approach: shift blame for losses from victims to perpetrators**
  - blame code executing while other threads are idle
  - blame code executed by lock holder when thread(s) are waiting
  - blame processes that arrive late to collectives
  - shift blame between CPU and GPU for hybrid code

# Directed Blame Shifting

- **Example:**
  - — **threads waiting at a lock are the symptom**
  - — **the cause is the lock holder**
- **Approach: blame lock waiting on lock holder**

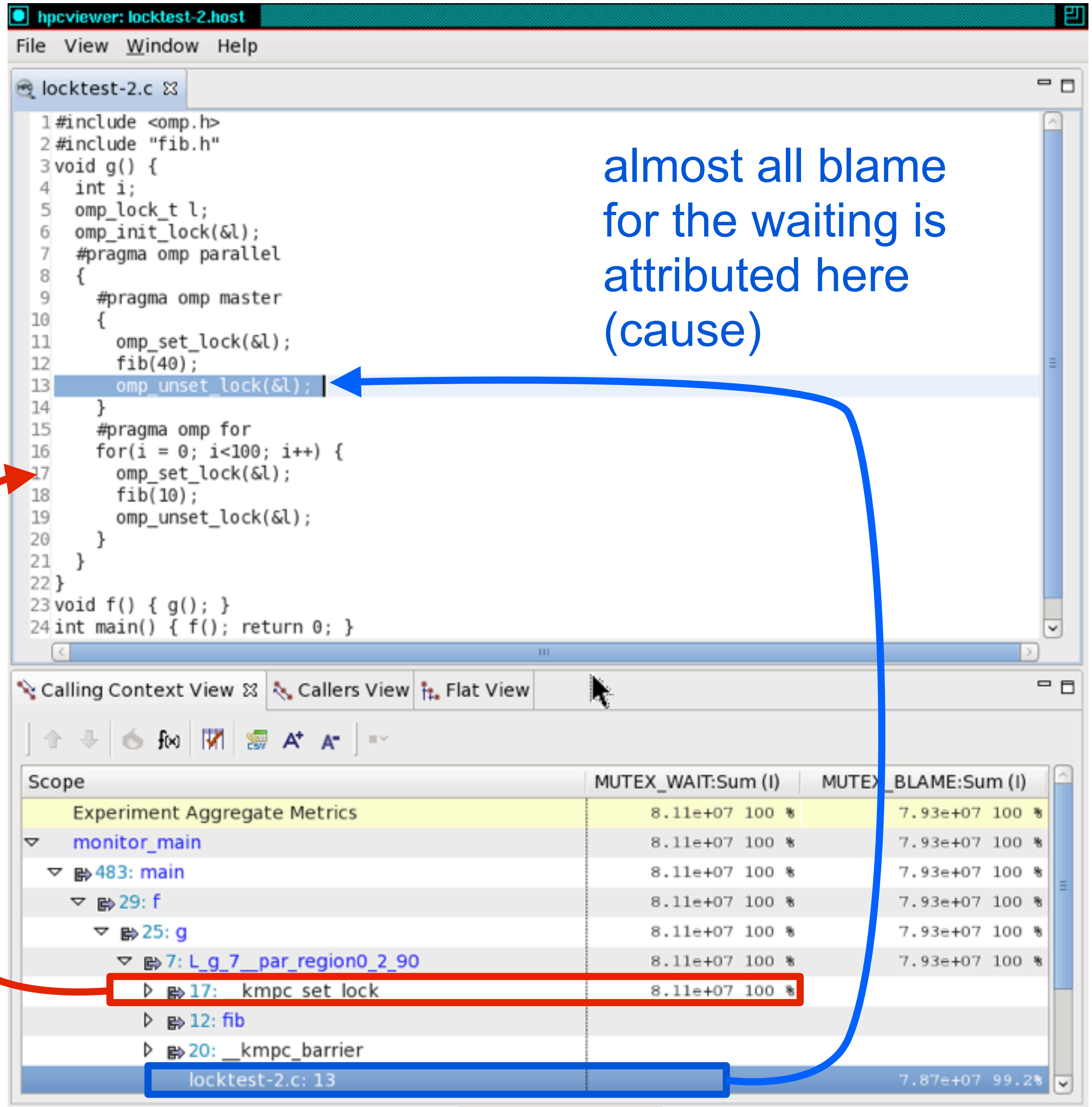


# Example: Directed Blame Shifting for Locks

**Blame a lock holder for delaying waiting threads**

- **Charge all samples that threads receive while awaiting a lock to the lock itself**
- **When releasing a lock, accept blame at the lock**

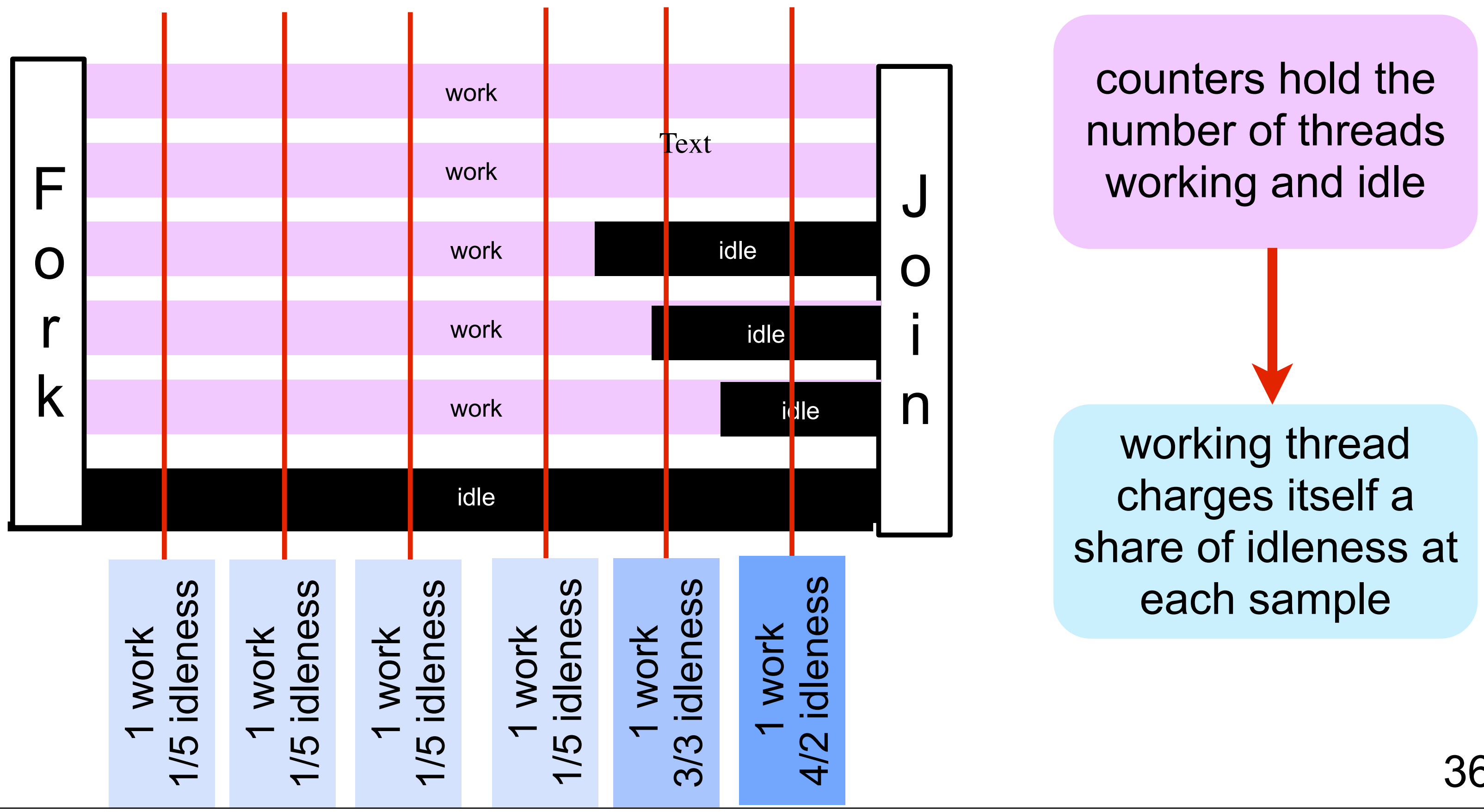
all of the waiting occurs here (symptom)



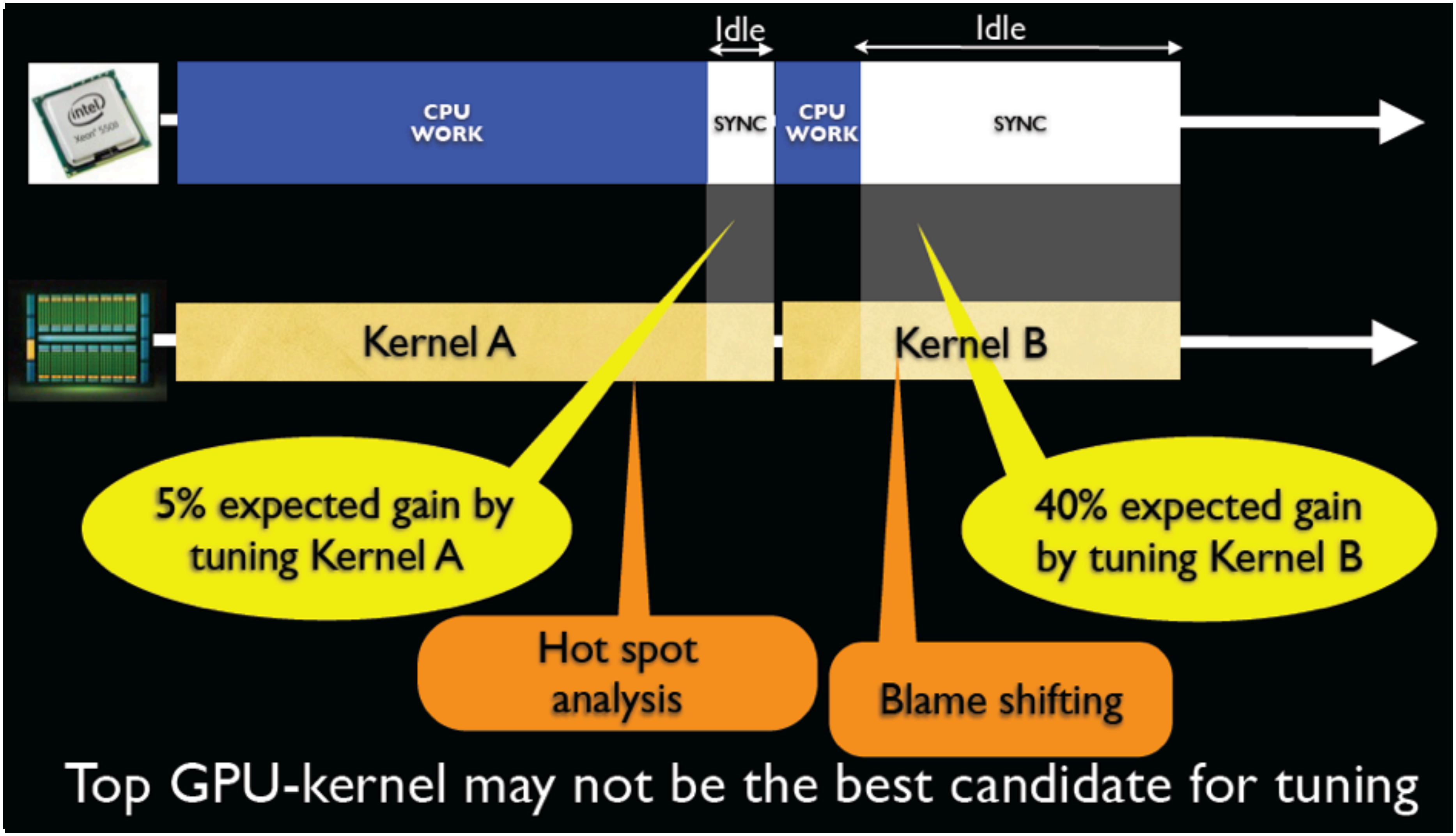

almost all blame for the waiting is attributed here (cause)

# Undirected Blame Shifting

- **Example:**
  - threads idling waiting for work are the symptom
  - the cause is insufficiently parallel work being executed by others
- **Approach: each working threads proportionally blames itself for instantaneous idling by others when it is sampled**



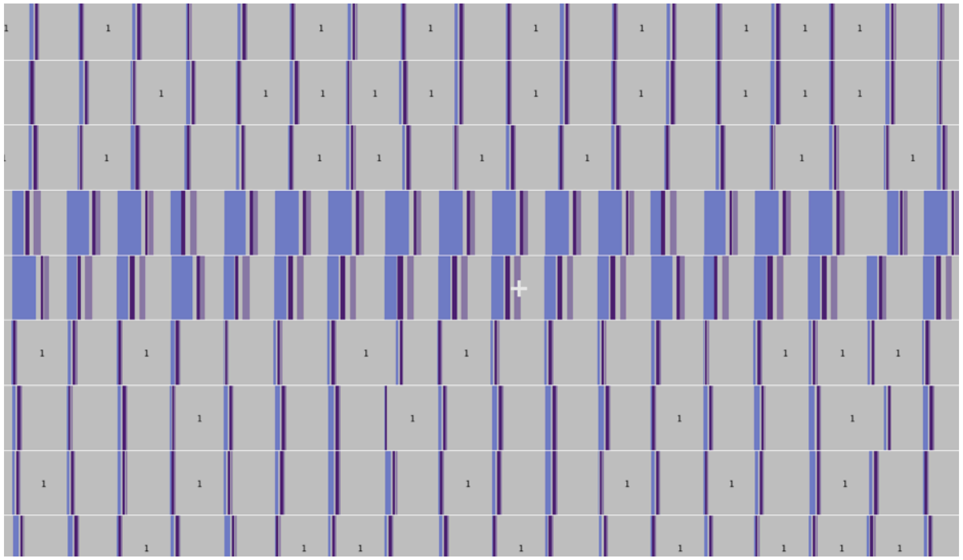
## Performance Expectations for Hybrid Code with Blame Shifting




Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. Effective Performance Tools for CPU/GPU Systems. SC13. To appear.

# GPU Successes with HPCToolkit

- **LAMMPS: identified hardware problem with Keeneland system**
  - **improperly seated GPUs were observed to have lower data copy bandwidth**



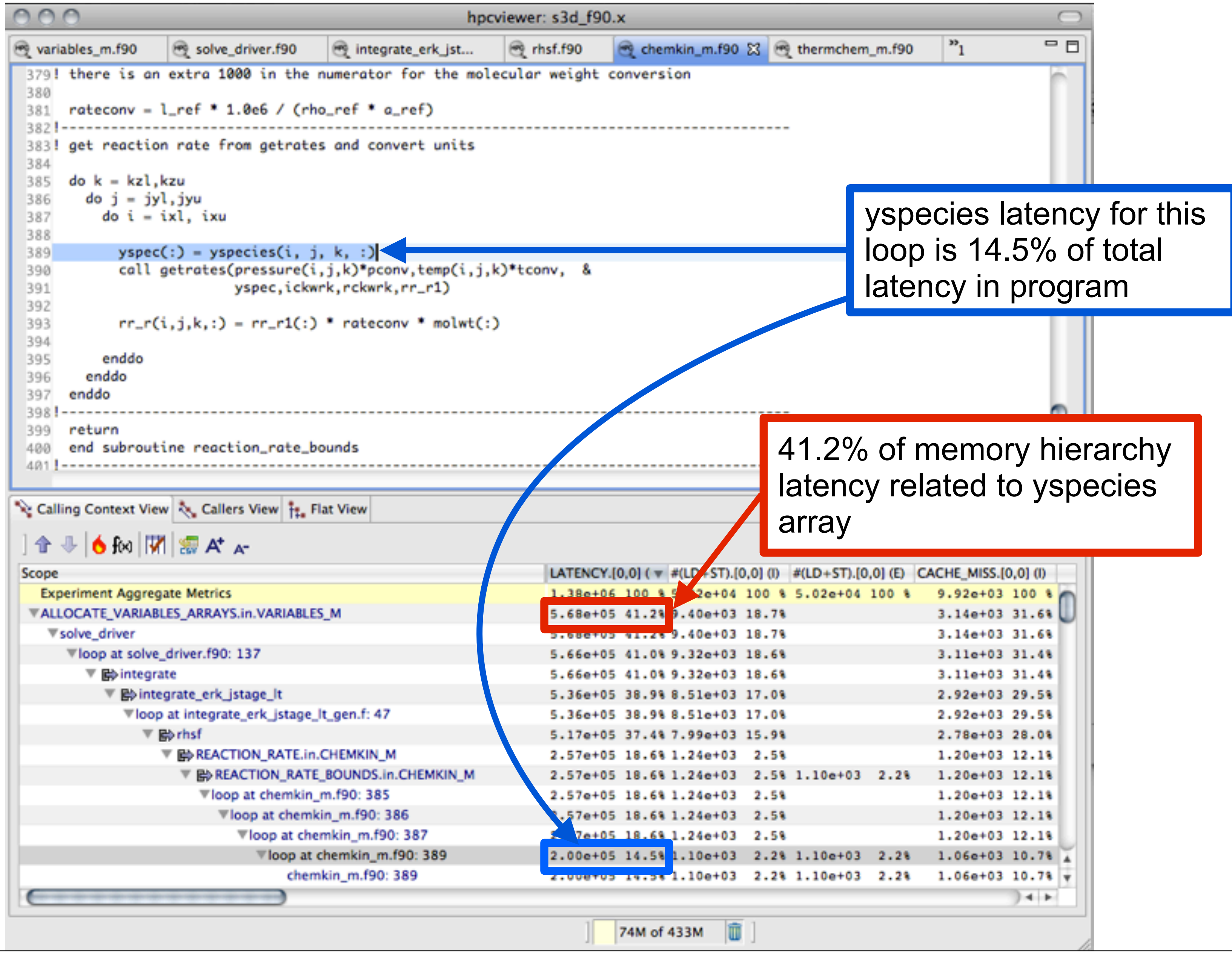
- **LLNL's LULESH: identified that dynamic memory allocation using cudaMalloc and cudaFree accounted for 90% of the idleness of the GPU**

# Data Centric Analysis

- **Goal: associate memory hierarchy performance losses with data**
- **Approach**
  - **intercept allocations to associate with their data ranges**
  - **measure latency with various PMU capabilities**
    - **instruction-based sampling (AMD Opteron)**
    - **precise event-based sampling + load latency facility (Intel)**
    - **marked instructions (IBM Power)**
  - **present quantitative results using hpcviewer**

Xu Liu and John Mellor-Crummey. A Data-centric Profiler for Parallel Programs. SC13. To appear.

# Data Centric Analysis of S3D



# Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  - — blame shifting
  - — attributing memory hierarchy costs to data
- Summary and challenges ahead

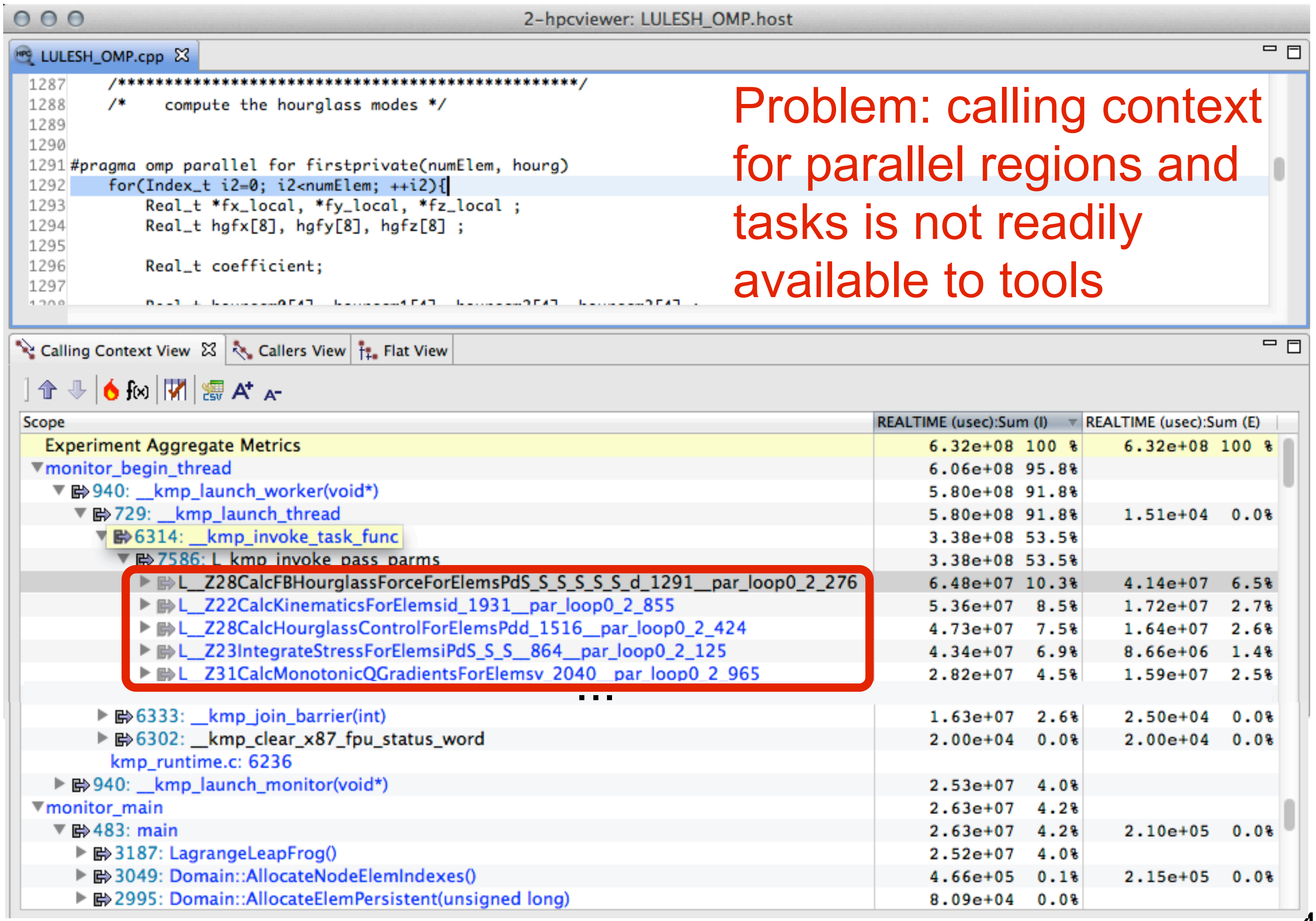
# Summary

- **Sampling provides low overhead measurement**
- **Call path profiling + binary analysis + blame shifting = insight**
  - scalability bottlenecks
  - where insufficient parallelism lurks
  - sources of lock contention
  - load imbalance
  - temporal dynamics
  - bottlenecks in hybrid code
  - problematic data structures
  - hardware counters for detailed diagnosis
- **Other capabilities**
  - attribute memory leaks back to their full calling context

# HPCToolkit Status

- Operational today on
  - 64- and 32-bit x86 systems running Linux (including Cray XT/E/K)
  - IBM Blue Gene
  - IBM Power7 systems running Linux
- Available as open source software at http://hpctoolkit.org
- Emerging capabilities
  - NVIDIA GPU
    - measurement and reporting using GPU hardware counters
  - data centric analysis
  - OpenMP analysis using OMPT

# OMPT: Emerging Monitoring for OpenMP

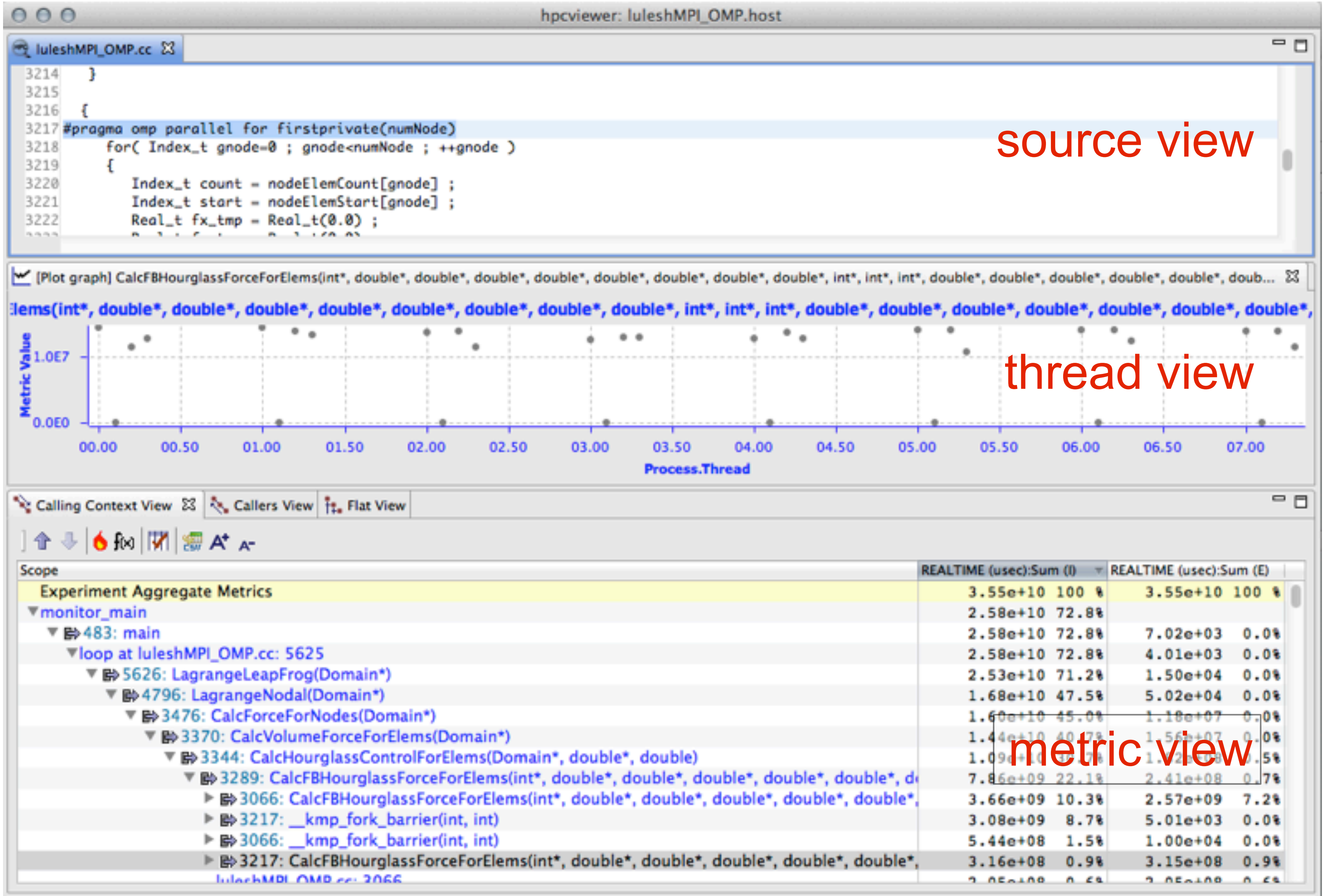


# Key OMPT Design Objectives

- **Enable tools to gather information and associate costs with application source and runtime system**
  - — provide interface for low-overhead sampling-based tools
  - — enable tools to reconstruct application-level profiles
    - – alternative to implementation-level view
  - — associate activity of a thread at any point in time with a state
    - – enable performance tools to monitor behavior
- **Negligible overhead if OMPT interface is not in use**
- **Define support for trace-based performance tools**

# Integrated View of MPI+OpenMP with OMPT

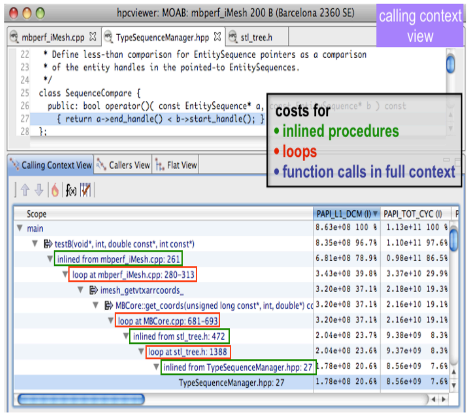
LLNL's luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000



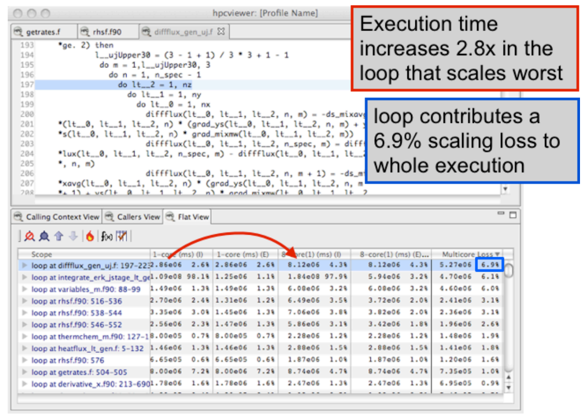
# Tool Challenges Ahead

- **Address challenges of emerging systems**
  - heterogeneity (e.g., on-chip; host + accelerator)
  - growth in thread counts: MIC supports 200+ threads
  - increasing scale of systems (e.g., Sequoia)
- **Identify <u>causes</u> rather than <u>symptoms</u> (blame shifting)**
- **Measure and analyze all facets of application performance**
  - CPU, accelerator, data movement, synchronization, I/O, power
  - interactions: HW, other jobs, system software
- **Analyze asynchronous activities**
- **Support dynamic adaptation of software**
  - measurements and decision algorithms to drive adaptation
  - assessment of adaptation policies
- **Provide analysis to support higher level insight, diagnosis, and guidance**

# HPCToolkit Capabilities at a Glance

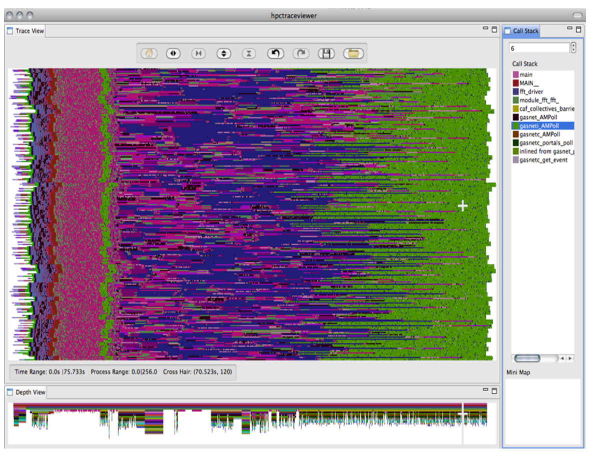



Attribute Costs to Code

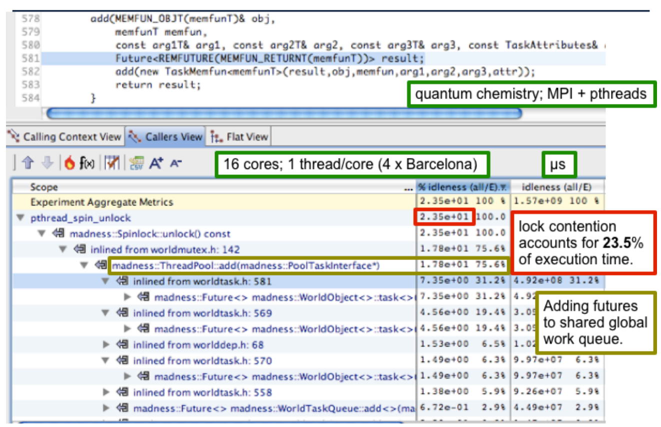



Pinpoint & Quantify Scaling Bottlenecks

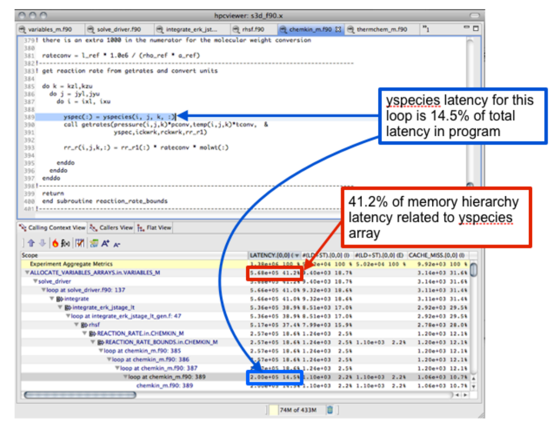
Assess Imbalance and Variability



Analyze Behavior over Time




Shift Blame from Symptoms to Causes




Associate Costs with Data

RICE

**hpctoolkit.org**